Disassembly Techniques for Reverse Engineering

Cedric Van Goethem
Ghent University
cedric.vangoethem@ugent.be

Djairho Geuens
Ghent University
djairho.geuens@ugent.be

Bart Middag
Ghent University
bart.middag@ugent.be

Abstract

Plenty of programs used today still come in the form of a binary executable. These binaries can be reverse engineered to retrieve structural and semantic information about their execution. By recreating assembly code from binary executables, disassemblers provide the first step toward this goal. To protect the intellectual property of binary programs, obfuscation techniques were invented, thwarting these disassemblers. In this paper, we provide a survey of the most common disassembly techniques and their reaction to the obfuscations. We show that most obfuscations can be undone and that future obfuscation techniques will have to find ways to thwart these intelligent disassemblers.

1. Introduction

In the process of reconstructing a program's source code from its binary executable, there are two consecutive steps to be taken. The first step is the regeneration of an assembly language program out of binary code. This step is called *disassembly*. When an assembly language program is available, the high-level source code can be reconstructed; this step is called *decompilation*. This survey paper focuses on the disassembly of binary executables.

Disassembly is especially useful when the original source code of a program is unavailable and modifications to the program have to be made. One can then translate the binary code into an assembly language program and modify it. Of course, disassembly can also be used for less legal activities, such as hacking.

To protect code, software developers started using obfuscation techniques [17, 18, 21, 23] to impede proper understanding. While previous research in the domain of obfuscation had been focused on the decompilation phase, multiple obfuscation techniques for binary code were developed by targetting weaknesses of disassemblers. Due to these developments, research in the domain of disassembly began to be focused on obfuscated code in the early 2000s.

Other papers on this topic often mix up research around successful disassembly of obfuscated binaries and the deobfuscation of these binaries. A clear distinction has to be made as these are two very different topics. With this overview of widely used obfuscation and disassembly techniques, we hope this paper will help provide a fresh insight into the current state of disassembly and the problems disassemblers must face.

2. Basic disassembly approaches

There are two main approaches to disassembling a binary executable, statically or dynamically.

The static approach analyzes the binary code without actually running it and produces a complete program containing all possible execution paths. It has the advantage of being able to analyze the entire file at once, but also has the disadvantage of being easier to mislead.

The dynamic approach analyzes the binary code by running it with a given input. This approach solves the problem that occurs in the static approach, but this also means that other possible execution paths than the one that is executed for the given input are not analyzed.

We can distinguish two main static disassembly techniques [25]: linear sweep and recursive traversal.
Linear sweep starts at the beginning of the binary file and decodes instructions sequentially. The algorithm does not take into account any semantical meaning of the disassembled instructions. Hence, data in code will also be decoded as instructions. Recursive traversal builds up the control flow graph of the program by following the control transfers it encounters. Each time a control transfer is identified, the target addresses of the control transfer are determined and the disassembler continues decoding instructions at the potential targets of the control transfer.

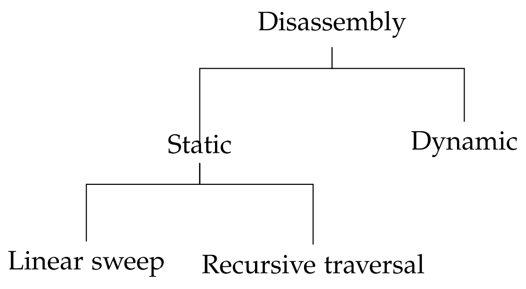

Figure 1: *General disassembly techniques*

3. Key problems in static disassembly

Standard disassemblers make the following assumptions [18, 14]:

3.1. *Linear sweep*

A0: All instructions are placed consecutively as bytes in the binary

The key problem is that data blocks and code blocks can be intermixed. This causes the linear sweep technique to interpret data blocks as code and wrongly recognize and extract instructions from it. This problem can be solved using recursive traversal.

3.2. *Recursive traversal*

A1: Conditional transfer instructions always have two possible targets: a fall-through address and a target address.

A2: Computed (indirect) control transfer instructions can be analyzed (e.g. jump tables)

A3: Function calls always return to the instruction right after the call instruction.

Some of these assumptions are not always completely justified. Assumption A1 can be mislead by creating a conditional transfer that always follows one path and then placing misleading instructions or junk bytes in the other path.

Also, assumption A2 is not always straightforward in the case of indirect, pointer-based jumps. In Figure 2, we illustrate indirect control transfers with a switch statement using a jump table (data section) inside the code. The selection of the case that has to be executed is performed using an indirect pointer based jump to an address in this jump table.

Furthermore, assumption A3 is not necessarily true and might cause a static disassembler to further examine data and falsely recognize it as code.

Another problem lies in the possibility of a program to modify its own instructions at runtime. This is called self-modifying code. When code modifies itself, a static analysis is useless for most of the time as it is not sure that the analyzed instructions will remain the same for different inputs.

4. Obfuscation

We can obfuscate the assembly code by modifying it so it does no longer satisfy the assumptions mentioned above. These modifications

```
9:  mov     eax,ebp                 #read index variable n
b:  sub     eax,0x1                 #index minus one
e:  cmp     eax,0x1                 #check index
11: ja      1a <default>            #jump if index > 2
13: jmp     DWORD PTR [eax*4+0x3c]  #jump to address depending on n
0000001a <default>: #default case
1a: mov     eax,ebp
1c: sub     eax,0x1
                                .
                                .
                                .
0000003c <table>:   #data block with jump table
3c: 44 00 00 00             dd      44 <C1>
40: 4b 00 00 00             dd      4b <C2>
00000044 <C1>:      #first case
44: bb 01 00 00 00          mov     ebx,0x1
49: eb f3                   jmp     36 <return>
0000004b <C2>:      #second case
4b: bb 01 00 00 00          mov     ebx,0x1
50: eb ec                   jmp     36 <return>
```

Figure 2: *x86 Assembly code for a Fibonacci program*

do not alter the flow of the program, but make it significantly harder to analyze.

4.1. *Opaque predicates*

To invalidate the assumption that each conditional transfer has two targets, we can modify an unconditional control transfer by introducing a condition that always evaluates to either true or false: an *opaque predicate*. This condition can be quite complex based on properties in linear algebra, calculus, and more. While its value is known at compile time, it will still have to be evaluated at runtime [4]. If the recursive traversal algorithm fails to recognize the opaque predicate, it will analyze both control paths.

Additionally, junk bytes can be placed in the path that will not be executed. On CISC instruction sets, the disassembly of these junk bytes can cause a misalignment of instructions which can propagate further into the valid instruction stream [18], causing instructions to be disassembled wrongly.[1]

This obfuscation can be undone by analyzing whether the value of such predicates either depends on the program's input or is dependent only on the internal program structure [5]. When the latter is observed, we can assume that program input has no influence on the execution of the program's flow and the unused control path can be ignored. This technique is called *use-dependence* tracking. By successfully recovering the unconditional jumps, recursive traversal can reconstruct a correct control flow graph using the original methods mentioned above.

Opaque predicates can also be detected by performing dynamic analysis. This will not reconstruct the whole assembly, but might instruct a static disassembler. During runtime, the dynamic disassembler might discover that some control paths are never taken. This information can be forwarded to the static disassembler and mark such predicates as potentially

[1]It can be shown that the instructions will resynchronize, following a distribution similar to the Kruskal Count [10].

opaque. This is called hybrid disassembly [19].

4.2. Branch functions

Branch functions are capable of completely hiding the control flow by invalidating the assumption that indirect control transfers can be analyzed.

This is done by replacing all control transfers by a call to a branch function as illustrated in Figure 3b. This function computes the target address based on an input value or the caller address. While the possible target addresses might be visible to some recursive traversal disassemblers, most disassemblers will not be able to identify the correct target address without executing the computation.

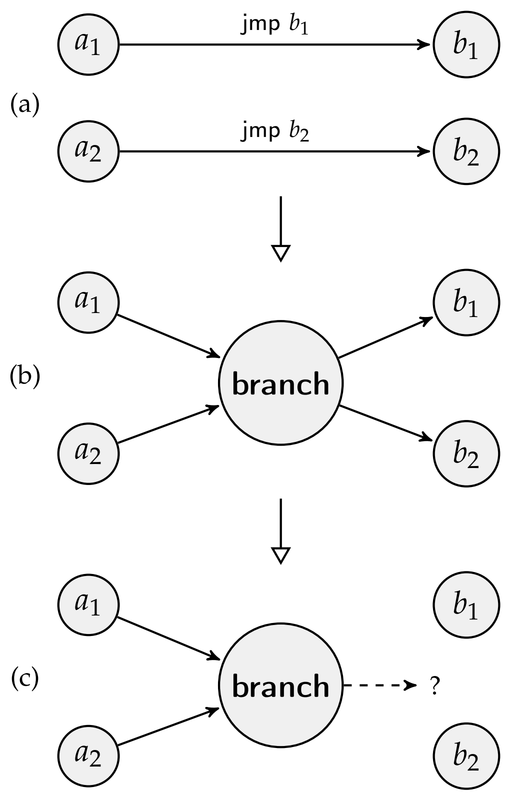

Figure 3: *Disassembled control flow in a branch function*

Additionally, it is possible the obfuscate the code of the branch function in such a way that a static disassembler will not be able to discover the set of potential targets of the branch function $(b_1, \ldots b_n)$. This is illustrated in Figure 3c.

By performing dynamic analysis, one can always correctly identify the target of a specific call to the branch function. However, as dynamic analysis takes time and may only execute part of the code, it is difficult to determine all possible targets. If one is only interested in modifying a part of the code, this method is however sufficient.[2]

4.3. Call stack tampering

As many disassemblers assume standard use of the `call` and `ret` instructions, obfuscation techniques often use these instructions to thwart disassemblers. Apart from non-returning calls, one can use *call stack tampering* for nonstandard control transfers with the `ret` instruction: pushing an address to the stack and returning is equivalent to an unconditional jump.

Currently, the only two ways to statically determine the true target of a suspicious `ret` instruction are `ret` target prediction [23] and use-dependence tracking for stack cells [15, 5]. Both methods work by analyzing past stack manipulations. If these methods fail, one has to resort to dynamic analysis to determine the target of the `ret` instruction.

4.4. Exception-based control transfers

Obscuring control flow can also be accomplished by using *exception-based control transfers*: it is very hard to statically predict which instructions might raise exceptions. Obfuscators can generate such instructions, which trigger a custom exception handler that, much like a branch function, computes the target address [21].

Static analysis will often fail to recognize these exception-based control transfers and as a result, the exception handlers and target blocks

[2]*Anti-debugging techniques* exist to greatly increase the cost of dynamic analysis and demotivate hackers. However, disassemblers like OllyDbg [28] and IDA Pro [24] can hide themselves from simple anti-debugging techniques.

might not be disassembled. However, it is simple to detect them with dynamic analysis: the debugger interface will identify all registered exception handlers when a fault occurs and inform the disassembler of any exception-raising instructions [23].

4.5. *Embedded virtual machines*

Many packers today use *embedded virtual machines* to hide sensitive code from hackers. An executable packed this way includes bytecode that cannot be executed without the code that implements the virtual machine.

While some disassemblers might correctly disassemble the instructions that implement the virtual machine, both static and dynamic disassemblers will falsely recognize the sensitive code as data. Many approaches to this problem work by first reverse-engineering the code that implements the virtual machine and then disassembling the bytecode. These outside-in approaches often make many assumptions about the virtual machine that are easily sidestepped.

However, Coogan and Debray [5] have proposed a novel method that disassembles the sensitive code inside-out by reconstructing the code based on the system calls the virtual machine makes.

4.6. *Self-modifying code*

As previously explained, static disassembly cannot handle *self-modifying code*, because it can only analyze the code before any modifications.[3] This can be used to obtain further resistance against static disassemblers. For example, many commercial software titles use encrypted code that is decrypted (hence generated) at runtime [23]. Decryption algorithms can range from a simple XOR operation to a sophisticated usage of cryptographic hash functions [1].

Using dynamic disassembly, it is possible to partially analyze the decrypted code. To analyze the full code, a hacker would have to manually undo the encryption and analyze the decrypted code. This holds true for the many other usages of self-modifying code.

5. Conclusion

Both disassembly and obfuscation are very active research domains. In the past decade, many other obfuscation techniques have been developed, like instruction overlapping [18, 10, 17, 16, 11]. These developments have inspired numerous reactions in the domain of disassembly [14, 27, 21, 13, 23]. Currently, research on disassembly has gained the upper hand with recent research by S. Debray [7], but in the future, more obfuscation techniques will be developed to counter this.

The challenge now lies in developing disassembly techniques that are resistant to all obfuscation techniques, even ones that have not been thought of yet. Such techniques would bring this cat-and-mouse game between the domains of obfuscation and disassembly to an end. However, it does not seem as if this is going to happen in the near future. It is likely that both domains will continue to be active and many more developments will be made on both sides in the years to come.

6. Acknowledgements

We would like to thank professor Koen De Bosschere for the enthousiastic and professional coaching during this semester. His advice was very constructive during the preparation of both the presentation and the paper.

We would also like to thank professor Joris Walraevens and dr. Dieter Claeys for the useful introduction they provided to this assignment. Over the semester, it has provided us with much-needed experience for scientific writing.

[3]While self-modifying code alone does not prevent disassembly completely, it impedes understanding, effectively limiting the usefulness of disassembly.

References

[1] J. Aycock, R. De Graaf, and M. Jr. Jacobson. Anti-disassembly using Cryptographic Hash Functions. *Journal in Computer Virology*, pages 79–85, 2006.

[2] C. Cifuentes and A. Fraboulet. Intraprocedural static slicing of binary executables. *Proceedings International Conference on Software Maintenance*, pages 188–195, 1997.

[3] C. Cifuentes and M. Van Emmerik. Recovery of Jump Table Case Statements from Binary Code. *Sci. Comput. Program.*, 40(2-3):171–188, 2001.

[4] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 184–196, New York, NY, USA, 1998. ACM.

[5] K. Coogan, G. Lu, and S. Debray. Deobfuscation of virtualization-obfuscated software: A semantics-based approach. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 275–284, New York, NY, USA, 2011. ACM.

[6] B. De Sutter, B. De Bus, K. De Bosschere, P. Keyngnaert, and B. Demoen. On the Static Analysis of Indirect Control Transfers in Binaries. In *PDPTA*, pages 1013–1019, 2000.

[7] S. Debray. Understanding software that doesn't want to be understood: Reverse engineering obfuscated binaries. `http://www.dagstuhl.de/mat/index.en.phtml?14241`, 2014. Accessed: November 12th 2014.

[8] R. N. Horspool and N. Marovac. An approach to the problem of detranslation of computer programs. *The Computer Journal*, 23(3):223–229, 1979.

[9] Barron Cornelius Housel, III. *A Study of Decompiling Machine Languages into High-level Machine Independent Languages*. PhD thesis, West Lafayette, IN, USA, 1973.

[10] M. Jacob, M. H. Jakubowski, and R. Venkatesan. Towards integral binary execution: Implementing oblivious hashing using overlapped instruction encodings. In *Proceedings of the 9th Workshop on Multimedia & Security*, pages 129–140, New York, NY, USA, 2007. ACM.

[11] C. Jamthagen, P. Lantz, and M. Hell. A new instruction overlapping technique for anti-disassembly and obfuscation of x86 binaries. In *Anti-malware Testing Research (WATeR), 2013 Workshop on*, pages 1–9, 2013.

[12] J. Kinder and H. Veith. Jakstab: A static analysis platform for binaries. In *Proceedings of the 20th International Conference on Computer Aided Verification*, pages 423–427, Berlin, Heidelberg, 2008.

[13] N. Krishnamoorthy, S. Debray, and K. Fligg. Static Detection of Disassembly Errors. *16th Working Conference on Reverse Engineering*, pages 259–268, 2009.

[14] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, pages 255–270, Berkeley, CA, USA, 2004. USENIX Association.

[15] A. Lakhotia, D. R. Boccardo, A. Singh, and A. Jr. Manacero. Context-sensitive Analysis of Obfuscated x86 Executables. In *Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '10, pages 131–140. ACM, 2010.

[16] C. LeDoux, M. Sharkey, B. Primeaux, and C. Miles. Instruction embedding for improved obfuscation. In *Proceedings of the 50th Annual Southeast Regional Conference*, pages 130–135, New York, NY, USA, 2012. ACM.

[17] B. Lee, Y. Kim, and J. Kim. binOb+: A Framework for Potent and Stealthy Binary Obfuscation. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, pages 271–281. ACM, 2010.

[18] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, pages 290–299. ACM, 2003.

[19] M. Madou, B. Anckaert, B. De Sutter, and K. De Bosschere. Hybrid static-dynamic attacks against software protection mechanisms. In *Proceedings of the 5th ACM Workshop on Digital Rights Management*, pages 75–82. ACM, 2005.

[20] S. Nanda, W. Li, L.C. Lam, and T. Chiueh. BIRD: Binary Interpretation Using Runtime Disassembly. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 358–370. IEEE Computer Society, 2006.

[21] I. Popov, S. Debray, and G. Andrews. Binary obfuscation using signals. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, pages 19:1–19:16, Berkeley, CA, USA, 2007. USENIX Association.

[22] N. Rosenblum, X. Zhu, B. Miller, and K. Hunt. Learning to Analyze Binary Computer Code. In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 2*, pages 798–804. AAAI Press, 2008.

[23] K. Roundy and B. Miller. Binary-code obfuscations in prevalent packer tools. *ACM Computer Survey*, 46(1):4:1–4:32, 2013.

[24] Hex-Rays SA. IDA Pro. `https://www.hex-rays.com/products/ida/`, 2014. Accessed: December 9th 2014.

[25] B. Schwarz, S. Debray, and G. Andrews. Disassembly of executable code revisited. In *Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, pages 45–55, Washington, DC, USA, 2002. IEEE Computer Society.

[26] H. Theiling. Extracting safe and precise control flow from binaries. In *Proceedings of the Seventh International Conference on Real-Time Systems and Applications*, pages 23–31, Washington, DC, USA, 2000. IEEE Computer Society.

[27] M. Venable, M. R. Chouchane, M. E. Karim, and A. Lakhotia. Analyzing memory accesses in obfuscated x86 executables. In *Proceedings of the Second International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 1–18, 2005.

[28] Oleh Yuschuk. OllyDbg. `http://www.ollydbg.de/`, 2014. Accessed: December 9th 2014.

7. Review report

This review report is about the following paper:
K. Coogan, G. Lu, and S. Debray. Deobfuscation of virtualization-obfuscated software: A semantics-based approach. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 275–284, New York, NY, USA, 2011. ACM

7.1. Summary

This paper proposes a novel technique for the deobfusction of virtualized obfuscated code. Virtual code is code based on an instruction set entirely generated by the obfuscator. The obfuscator translates the original assembly code into its virtual equivalent. At runtime this code is interpreted by a virtual machine embedded in the executable. The goal of this paper is to distinguish the virtual machine from the actual instructions and to reconstruct the original assembly code.

The key problem with virtual code analysis is that there is no prior knowledge about the instructions and control transfers, due to the randomization of instruction set. Therefor, not many assumptions can be made. In this paper, the assumption is made that system calls can be correctly identified since they have to meet the ABI calling conventions. Using dynamic analysis, the algorithm tries to compute the origin of the system call arguments, which can be inferred from the ABI specification, and the instructions that indirectly influence their value. This is a novel technique called use-dependence tracking. From these dependence chains, a distinction can be made between instructions that directly influence system calls and those who don't. The former have a high chance to be the original assembly code and the latter are most likely the virtual machine's interpreter code.

By making only one assumption about the system calls, this technique can be seen as a new general approach towards the deobfuscation of virtualized code. From their results, it is shown that their algorithm can correctly distinguish the virtual machine from the original code in most cases. It states that their technique defeats most of the current obfuscation techniques, but can be easily thwarted by introducing false dependences for the system calls.

7.2. Review

This review format was based on the ACM TACO Journal review guidelines.

General opinion: This paper solves a complex problem in the field of deobfuscation of virtualized code. It introduces a novel technique which might serve as a seminal paper for further research. The article is well written and the concepts are clearly explained in each section. The results are promising, but the choice of test cases is a bit poor. Further tests on larger binaries would be an interesting addition to the paper or a next publication.

Does this paper present innovative ideas or material? Yes, such as:

- Use dependence-analysis
- A unified approach for virtual code analysis
- Filtering virtualized code from the original assembly

Is the information in the paper sound, factual, and accurate? Yes.

What are the major contributions of the paper? It proposes a virtual machine independent technique for deobufscation of virtual machine obfuscated binaries.

Does this paper cite and use appropriate references? Yes.

Is the treatment of the subject complete? Yes, all relevant control flow obfuscations are addressed. Therefor, this technique can be applied to real obfuscated binaries.

Recommendations to the authors:

- We suggest writing automated test cases so the analysis does not have to be done by hand. It would be interesting to see analysis on larger real world applications instead of hand-crafted toy programs.
- The reason why some false positives occur during the analysis isn't quite clear. Is this because of intrinsic properties of this technique, or does it need further optimisations? This could use some further explanation as a guidance for future research.
- Some more information towards further work that could be helpful to defeat the proposed obfuscation in section 4.
- Publication of the source code and how to obtain it would be helpful for further research.

Recommendation to the publisher: Accept.